Serveur Web Répliqué

Mémoire de diplôme, Automne et Hiver 1999/2000

Yann Schluchter

Responsable : Matthias Wiesmann

Laboratoire de Systèmes d'Exploitation
Ecole Polytechnique Fédérale de Lausanne

18 février 2000

Table des matières

Table des figures

Liste des tableaux

Avant-propos

Remerciements

Conventions typographiques

Ce document utilise les conventions typographiques décrites dans le tableau 1.

Font	Description
Sans serif	Schémas, figures
`Courrier`	Codes, algorithmes
Italique	Termes anglais

TAB. 1 – Convention typographiques

Glossaire

API ***Application Programming Interface*** Ensemble de bibliothèques et fonctions fournies avec un programme permettant d'utiliser son potentiel et de greffer de nouvelles fonctions en programmant des modules externes à partir d'un langage donné.

Applet Composant logiciel écrit en Java s'exécutant au sein du navigateur du client.

ASP ***Active Server Page*** Macro-langage de script conçu par MicroSoft, qui se charge d'interpréter depuis un serveur HTTP où il est installé certains mots-clés inclus dans des fichiers au format HTML mais d'extension .ASP.

CORBA ***Common Object Request Broker Architecture*** Document de synthèse décrivant les mécanismes d'échange entre les modules d'applications développés pour une architecture distribuée. Cette norme a vu le jour en 1991 sous le patronage de l'O.M.G. (voir chapitre 4, page 23).

CGI ***Common Gateway Interface*** Norme d'interfaçage d'applications exécutables et de sites Web. Cette norme permet de mettre à disposition des concepteurs de pages, d'outils tels que compteurs de visite, traitement de formulaires, gestion de bases de données, etc...
Ces outils logiciels sont exécutés automatiquement par le serveur lorsque

qu'un lien sur une page du site y fait référence, et renvoie le résultat du traitement à l'emplacement du lien dans la dite page.

HTML ***HyperText Markup Language*** Langage de description de pages consultables sur écran, qui permet par l'emploi de codes textuels (balises/tags) dans le corps même du document d'indiquer à l'application qui l'affichera, à quoi correspond telle portion de texte (entête, corps, note, titre, texte...).

Ce langage qui est dérivé de SGML offre en plus la possibilité de créer des liens hyper-textes.

HTTP ***HyperText Transfert Protocol*** Protocole de communication utilisé pour le transfert entre un serveur et un poste client des pages au format HTML des sites WEB (voir chapitre 2, page 11).

OGS Description de OGS (voir chapitre 5, page 27).

Servlet Composant logiciel écrit en Java s'exécutant du côté du serveur en réponse à des requêtes (voir chapitre 3, page 17).

URL ***Uniform Resource Locator*** Ligne de commande décrivant une ressource Internet (serveur, fichier, groupe de discussion, etc) et le moyen d'y accéder.

Ex : http ://www.nono.com/present.html indique d'aller cherche la page present.html du serveur www.nono.com à l'aide du protocole de transfert HTPP.

Ex : ftp ://psg.com indique d'accéder au serveur de fichiers à l'aide du protocole de communication et transfert FTP. Voir utilement : RFC 1738.

Chapitre 1

Introduction

Le but de ce projet est de construire un serveur web répliqué en utilisant le service de réplication CORBA OGS. L'approche envisagée n'est pas de répliquer le serveur web en entier, mais de ne répliquer que le service de servlets. Celles-ci permettent d'implémenter différents services, y compris les services de serveur de fichier.

1.1 Contexte

Le succès grandissant d'Internet a fait apparaître, il y a quelques années, le besoin de répliquer certains serveurs web. A cela deux raisons. La première, c'est l'optimisation des performances ; si plusieurs serveurs se partagent le travail, les temps de réponse seront plus courts. La deuxième, c'est la répartition géographique du trafic ; des serveurs web correctement répartis géographiquement ne créent pas de concentration de trafic.

Une motivation plus récente pour faire de la réplication de serveurs web est la tolérance aux fautes. Certaines compagnies sont très présentes sur Internet et il est impératif pour elles que leur site web soit en tout temps accessible.

1.1.1 Réplication de serveurs web

Les travaux qui ont été fait sur la réplication de serveurs web concernent principalement le reroutage des requêtes HTTP. En effet, un client ne s'adresse qu'à une URL et un algorithme doit rerouter la requête sur un des répliqua du site web. A ce jour, plusieurs algorithmes existent. Ils sont basés sur des critères de performances ou de géographie. Dans ce projet, nous n'étudierons pas ce problème, étant donné que cela a déjà été fait.

Deux types de réplication sont envisageables. La réplication en "lecture seule" et la réplication en "lecture/écriture".

Réplication en "lecture seule"

La réplication en "lecture seule" (*mirroring*) est ce qui se fait principalement aujourd'hui. Le principe est simple : le contenu du site web est publié sur différentes machines correctement réparties géographiquement et un serveur se charge de rerouter les requêtes vers ces machines.

Les avantages d'une telle réplication sont les suivants :

- Diminution du temps de réponse
- Meilleure répartition du trafic
- "Tolérance aux fautes"

L'inconvénient de cette méthode de réplication est que les données fournies par un client ne sont pas mises à jour sur les autres serveurs. Cela peut poser des problèmes si les serveurs doivent travailler avec des données communes (incohérence des données fournies par les clients). Egalement, en cas de *crash* d'un des serveurs, on risque de perdre des données.

Réplication en "lecture/écriture"

La réplication en "lecture/écriture" est en tous points identique à la réplication en "lecture seule", sauf qu'il existe un mécanisme de mise à jour des données fournies par les clients sur les différents serveurs.

Les avantages sont les mêmes qu'avec la réplication en "lecture seule" avec en plus la cohérence des données sur les différents serveurs web. De plus, si un serveur tombe en panne, quand il est réparé et remis en service, sa mise à jour se fait automatiquement en fonction de l'état des autres serveurs.

Un inconvénient possible d'une telle réplication peut être une augmentation du temps de réponse pour les requêtes qui modifient l'état du serveur, car la réplication de celles-ci sur les différents serveurs risque de prendre un peu plus de temps. Mais pour les requêtes qui ne font que lire des données sur le serveur, cela ne change rien.

1.2 Environnement

L'environnement de développement sera le suivant :

- Java : Version 1.1.6 from Sun Microsystems Inc.
- Serveur de servlets : JSDK 2.1 from Sun Microsystems Inc.
- VisiBroker Developer for Java [03.02.02.C1.01] (MAY 01 1998 16 :45 :39)
- OGS version ? ? pour Visibroker
- OS : Solaris version 2.x ; CPU : sparc

Chapitre 2

HTTP

Le protocole HTTP (*Hypertext Transfer Protocol*) est un protocole basé sur TCP/IP. Il définit d'une manière précise la manière dont les clients communiquent avec les serveurs web. HTTP/1.0 [6] est le protocole le plus utilisé aujourd'hui. Mais son successeur HTTP/1.1 [5] est en train de le remplacer.

2.1 Bases du protocole HTTP

Le protocole HTTP suit un simple paradigme client/serveur. En résumé, un échange entre un navigateur (client) et un serveur web se fait de la manière suivante : le client ouvre une connection avec le serveur (généralement sur le port 80) et lui envoie une requête. Le serveur répond à la requête et la connection est fermée[1] (voir figure 2.1).

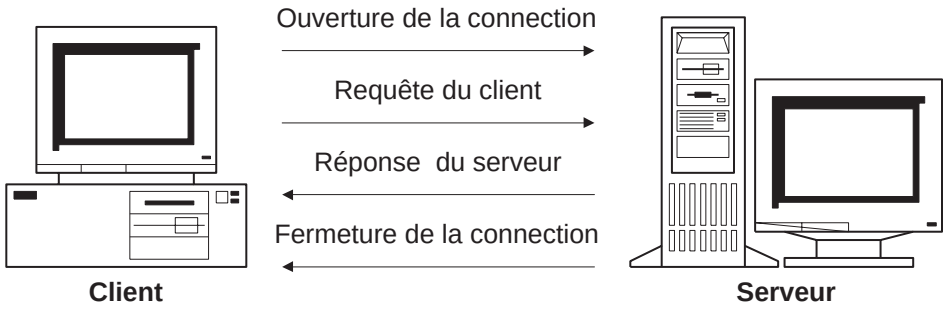

FIG. 2.1 – Nature client/serveur du protocole HTTP

2.1.1 Protocole sans connection

Le protocole HTTP est dit sans connection (*connectionless*). Avec un tel protocole, le client ouvre une connection avec le serveur, envoie une requête, reçoit une réponse et ferme la connection. Contrairement aux protocoles orientés

[1] Dans le protocole HTTP/1.1, on peut garder la connection ouverte pour optimiser les requêtes (entête *keep-alive*).

connection où le client se connecte au serveur, envoie une requête, reçoit une réponse et garde la connection ouverte pour pouvoir éventuellement envoyer d'autres reqêtes.

La nature sans connection de HTTP est à la fois un avantage et un inconvénient. L'avantage, c'est qu'une connection courte fait que le serveur n'est pas longtemps monopolisé par une requête et peut donc servir un plus grand nombre de clients à la fois. L'inconvénient, c'est qu'à chaque requête une connection doit être établie, ce qui implique un temps de réponse plus long.

2.1.2 Protocole sans état

Le protocole HTTP est aussi dit sans état (*stateless*). Un protocole est dit sans état s'il n'a pas connaissance des connections précédentes et qu'il ne peut pas faire la différence entre une requête et une autre.

Les avantages d'un protocole sans connection, c'est que le protocole est plus simple et consomme moins de ressources sur le serveur. De plus, en cas de *crash*, la reprise est plus facile (pas de *handshaking*). L'inconvénient, c'est qu'il ne permet pas de suivre un client lorsqu'il parcourt un site web.

2.2 Structure d'une requête HTTP

Une requête HTTP peut se décomposer en cinq parties distinctes : la méthode, le nom de la ressource, le protocole, les entêtes et le corps de la requête. Sur l'exemple de la figure 2.2, GET est la méthode, /index.html est le nom de la ressource et HTTP/1.0 est le protocole. Le reste, ce sont des entêtes. Il n'y a pas de corps pour cette requête.

```
GET /index.html HTTP/1.0
User-Agent: Mozilla/4.02 [en] (Win95; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, */*
```

FIG. 2.2 – Exemple de requête HTTP

Une requête HTTP se termine toujours par deux retours de chariot et sauts de ligne (CRLF).

2.2.1 Méthodes

HTTP/1.0 définit trois méthodes : GET, POST et HEAD. Il existe encore quatre autres méthodes, mais elles ne sont pas définies dans les spécifications de HTTP/1.0. Il s'agit des méthodes PUT, DELETE, LINK et UNLINK.

Méthode GET

La méthode GET est la requête la plus commune. Elle est utilisée pour demander de l'information au serveur. A chaque fois qu'un navigateur demande une page à un serveur web, il utilise la méthode GET. Le nom de la ressource de la requête correspond au fichier qui est demandé.

Une requête qui utilise la méthode GET ne doit en aucun cas modifier l'état des données sur le serveur. Cette règle est souvent transgressée par les développeurs qui utilisent la méthode GET pour envoyer un formulaire HTML. Si l'application qui reçoit les données d'un tel formulaire doit modifier l'état des données sur le serveur, il faut utiliser la méthode POST.

Lors d'une requête utilisant la méthode GET, les éventuelles données qui doivent êtres transmises au serveur (par exemple les champs d'un formulaire HTML) sont passées dans le nom de la ressource de la requête. Les données seront séparées du nom de la ressource par le caractère '?' et les différentes paires nom=valeur seront séparés par le caractère '&', comme on peut le voir sur la figure 2.3. Sur le serveur, ces données (appelées *query string*) sont généralement stockées dans une variable d'environnement. Lorsqu'on veut transmettre beaucoup de données, on évitera cette méthode, car la taille fixe des variables d'environnement peuvent tronquer les données.

```
GET /cgi-bin/login.cgi?username=Dupont&password=dup99 HTTP/1.0
User-Agent: Mozilla/4.02 [en] (Win95; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, */*
```

FIG. 2.3 – Exemple de requête HTTP utilisant la méthode GET

Méthode POST

La méthode POST est généralement utilisée pour passer des données entrées par le client dans un formulaire HTML au serveur. La grande différence avec la méthode GET, c'est que les données ne sont pas passées dans le nom de la ressource de la requête, mais dans le corps de la requête qui vient juste après les entêtes (voir figure 2.4). Sur le serveur, ces données arriveront sur l'entrée standard de l'application qui doit les traiter, il n'y aura donc pas de limitation quand à la taille de ces données.

Une autre différence avec la méthode GET, c'est que l'application qui traite une requête utilisant la méthode POST peut modifier l'état des données sur le serveur.

Méthode HEAD

La méthode HEAD est identique à la méthode GET sauf que la réponse ne contiendra que l'entête de la réponse et pas le corps. Cette méthode est utilisée

```
POST /cgi-bin/login.cgi HTTP/1.0
User-Agent: Mozilla/4.02 [en] (Win95; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, */*
Content-Length: 30

username=Dupont&password=dup99
```

FIG. 2.4 – Exemple de requête HTTP utilisant la méthode POST

pour faire du deverminage, vérifier des liens hypertextes ou pour vérifier si un fichier existe avant d'essayer de le recevoir. Cette méthode nous permet aussi d'obtenir la date de la dernière modification d'un fichier pour savoir si on doit le charger ou si on peut utiliser celui qui se trouve éventuellement dans notre cache.

2.2.2 Entêtes

Les entêtes transmises dans une requête HTTP peuvent contenir toutes sortes de données. Elles ont le format suivant :

<nom de l'entête> :<valeur de l'entête>CRLF.

Les entêtes les plus courantes sont :

User-Agent Décrit le navigateur utilisé par le client.

Accept Liste des format de fichiers que le client peut accepter.

Content-Length Taille des données contenues dans le corps de la requête.

Content-Type Type des données contenues dans le corps de la requête (format MIME).

2.3 Structure d'une réponse HTTP

Une réponse HTTP peut aussi se décomposer en 5 parties : Le protocole, le code de réponse, le message du code de réponse, les entêtes et le corps de la réponse. Sur l'exemple de la figure 2.5, HTTP/1.0 est le protocole, 200 est le code de la réponse, OK est le message du code de réponse. Les trois lignes suivantes sont les entêtes et le reste est le corps de la réponse.

Le code de réponse donne une information sur le déroulement de la requête. Le tableau 2.1 nous donne une indication sur la signification de ces codes. Le message accompagnant le code de réponse n'est là que par souci de lisibilité (pour les développeurs).

```
HTTP/1.0 200 OK
Server: Netscape-Enterprise/2.01
Content-Type: text/html
Content-Length:93

<html>
<head>
<title>Hello World Page !</title>
</head>
<body>
Hello World !
</body>
</html>
```

FIG. 2.5 – Exemple de réponse HTTP

Code	Carégorie	Description
1xx	Information	Code provisoir utilisé pour des applications experimentales.
2xx	Succès	La requête à correctement été reçue, comprise et acceptée.
3xx	Redirection	Le serveur demande au client de se rediriger vers une autre URL.
4xx	Erreur du client	La requête n'est pas bien formattée ou ne peut pas être réalisée.
5xx	Erreur du serveur	Une requête valide a été reçue, mais le serveur n'a pas pu la réaliser.

TAB. 2.1 – Codes de retour HTTP

Les entêtes contiennent diverses informations sur le serveur et sur le contenu du corps de la réponse. Les plus courantes pour les réponses sont :

Server Décrit le logiciel utilisé par le serveur.

Content-Length Taille des données contenues dans le corps de la réponse.

Content-Type Type des données contenues dans le corps de la réponse (format MIME).

Le corps de la réponse contient généralement le contenu du fichier qui a été demandé dans la requête (par exemple une page HTML ou une image au format GIF ou JPEG).

Chapitre 3

Servlets

3.1 Introduction

En combinaison avec les formulaires HTML, les CGI fournissent un mécanisme d'interaction entre le client et le serveur. Cette technologie a ouvert le chemin à de nouvelles idées comme le support à la clientèle en ligne ou le commerce électronique. Mais la technologie des CGI commence à dater. Le manque de performances et de flexibilité se ressent de plus en plus. De nouvelles technologies doivent êtres trouvées pour remplacer les CGI. Parmi les technologies existantes (ASP, Servlets, etc.) les servlets [1] paraissent être un digne successeur des CGI pour le développement d'applications basées sur le Web.

3.2 Qu'est-ce qu'une servlet ?

Une servlet est un composant logiciel écrit en java qui se trouve du côté du serveur et qui étend les fonctionnalités de ce dernier. De même qu'une applet est exécutée dans le navigateur du client, la servlet s'exécute au sein du serveur. Par contre, contrairement aux applets, les servlets ne peuvent pas afficher directement une interface graphique à l'utilisateur. Une servlet est exécutée en tâche de fond sur le serveur et seul le résultat est retourné au client (généralement sous forme de code HTML).

Plus précisément, les servlets sont les classes Java conformes à une interface spécifique qui peuvent êtres invoquées sur un serveur. Il faut noter que les fonctionnalités proposées par les servlets ne sont pas limitées aux serveurs Web. N'importe quel serveur qui supporte l'API des servlets peut bénéficier des avantages fournis par les servlets, par exemple, les serveurs FTP, Telnet, mail et news. L'API des servlets est une spécification développée par Sun qui définit les classes et les interfaces utilisées pour créer et exécuter les servlets.

Les servlets fournissent un environnement de travail pour créer des applications qui implémentent le paradigme client/serveur. Par exemple, quand un navigateur envoie une requête au serveur, celui-ci fait suivre la requête à une servlet qui la traite et qui renvoie une réponse (généralement au format HTML)

qui est transmise au client.

Les servlets ont de multiples avantages sur les autres technologies existantes. En voici quelques uns :

- Utilisation du langage de programmation Java (langage orienté objet, indépendant de la plateforme, compilation)
- Exécution dans des *thread* (initialisation moins coûteuse qu'un processus)
- Compatible avec la plupart des serveurs web (Apache, Netscape Enterprise, Java Web Server, etc.)

3.3 Structure de base des servlets

Les servlets sont de simples applications Java, à la différence prêt qu'elles héritent de classes spéciales et qu'elles implémentent certaines méthodes de ces classes. Elles s'écrivent aussi facilement qu'un programme Java.

Pratiquement, toutes les servlets ont deux choses en commun. La première, c'est qu'elle héritent toutes soit de la classe `Servlet`, soit de la classe `HttpServlet`[1]. La seconde, c'est que toutes les servlets réimplémentent au moins une méthode dans laquelle les fonctionnalités sont décrites.

Voyons maintenant le squelette d'une servlet type. Par souci de clareté, les paramètres des différentes méthodes ont été omis.

```
1  public class ServletSquelette extends HttpServlet {
2
3    public void init() {
4
5      // Code d'initialisation
6    }
7
8    public void service() {
9
10     // Code d'execution de la servlet
11   }
12
13   public void destroy() {
14
15     // Liberation des ressources
16
17   }
18 }
```

Listing 3.1 - Classe ServletSquelette

[1] La classe `HttpServlet` hérite elle-même de la classe `Servlet`

3.4 Développement des servlets

Une servlet HTTP hérite de la classe `HttpServlet`. Pour être fonctionnelle, elle doit implémenter au moins une des méthodes décrites dans le tableau 3.1.

Méthode	Description
`service()`	Le serveur appelle cette méthode quand il reçoit une nouvelle requête. Si cette méthode n'est pas réimplémentée, son rôle est d'appeler une des méthodes ci-dessous en fonction de la méthode utilisée par la requête HTTP. Si elle est réimplémentée, ces méthodes ne sont plus appelées automatiquement.
`doGet()`	Cette méthode est appelée en réponse à une requête HTTP utilisant la méthode GET. Cette méthode est également appelée si la méthode `doHead()` est appelée (si celle-ci n'est pas réimplémentée). Si la servlet doit pouvoir répondre à des requêtes GET, cette méthode doit être implémentée.
`doPost()`	Cette méthode est appelée en réponse à une requête HTTP utilisant la méthode POST. Si la servlet doit pouvoir répondre à des requêtes POST, cette méthode doit être implémentée.
`doHead()`	Cette méthode est appelée en réponse à une requête HTTP utilisant la méthode HEAD. Si elle n'est pas réimplémentée, son rôle sera d'appeler la méthode `doGet()` et de retourner l'entête HTTP générée par celle-ci.

TAB. 3.1 – Méthodes nécessaires pour créer une servlet

Il existe encore 5 autres méthodes que l'on peut réimplémenter (`doPut()`, `doDelete()`, `doTrace()`, `doOptions()` et `getLastModified()`), mais nous ne les verrons pas en détail.

Les méthodes décrites dans le tableau 3.2 ne sont pas indispensables, mais elles sont généralement implémentées si l'on veux écrire une servlet "propre".

Méthode	Description
`init()`	Cette méthode est appelée une seule fois lors du chargement de la servlet. Elle est utilisée pour initialiser les ressources qui sont utilisées par la servlet (par exemple une connection sur une base de données).
`destroy()`	Cette méthode est appelée lors du déchargement de la servlet par le serveur. Elle est utilisée pour libérer les ressources utilisées par la servlet.
`getServletInfo()`	Cette méthode permet d'identifier la servlet. Elle peut être appelée par le serveur pour connaître les servlets qui sont chargées à un instant donné

TAB. 3.2 – Méthodes communément implémentées

3.5 Cycle de vie des servlets

Le cycle de vie d'une servlet peut être décomposé en neuf points que nous allons voir.

1. Le serveur charge la servlet quand celle-ci est demandée par un client ou au démarrage du serveur de servlet si celui-ci est configuré comme cela. Cette action est équivalente au code Java suivant :

   ```
   Class c = Class.forName("com.sourcestream.MyServlet");
   ```

   Il faut noter que lorsqu'on parle de servlet, le terme charger désigne souvent le processus de chargement et d'instanciation de la servlet.

2. Le serveur crée une ou plusieurs instances de la classe de la servlet. Cela dépendra de l'implémentation de la servlet. Cette action correspond au code Java suivant :

   ```
   Servlet s = (Servlet) c.newInstance();
   ```

3. Le serveur construit un objet `ServletConfig` qui contient les informations pour l'initialisation de la servlet.
4. Le serveur appelle la méthode `init()` de la servlet en lui passant en paramètre l'objet `ServletConfig` créé au point précédent. On a la garantie que l'exécution de la méthode `init()` sera terminée avant que la première requête n'arrive à la servlet.
5. Le serveur crée un objet `ServletRequest` ou `HttpServletRequest` à partir des informations contenues dans la requête. Il crée aussi un objet `ServletResponse` ou `HttpServletResponse` qui fournit des méthodes pour extraire la réponse de la servlet. Le type d'objet créé dépend du fait que la servlet hérite de la classe `servlet` ou de la classe `HttpServlet`.
6. Le serveur appelle la méthode `service()` de la servlet en lui passant en paramètres les deux objets créés au point précédent. Quand des requêtes concurrentes arrivent, plusieurs méthodes `service()` peuvent tourner en même temps dans des *threads* séparés.
7. La méthode `service()` traite la requête du client en évaluant l'objet `ServletRequest` ou `HttpServletRequest` et fournit une réponse en utilisant l'objet `ServletResponse` ou `HttpServletResponse`.
8. Si le serveur reçoit une autre requête pour cette servlet, le processus recommence au point 5.
9. Quand le serveur doit décharger la servlet, il appelle la méthode `destroy()`.

On peut retrouver ces neuf points sur la figure 3.1 qui illustre le cycle de vie d'une servlet.

3.6 Avantages des servlets sur les CGI

Les trois principaux avantages des servlets sur les CGI sont la performance, la portabilité et la sécurité.

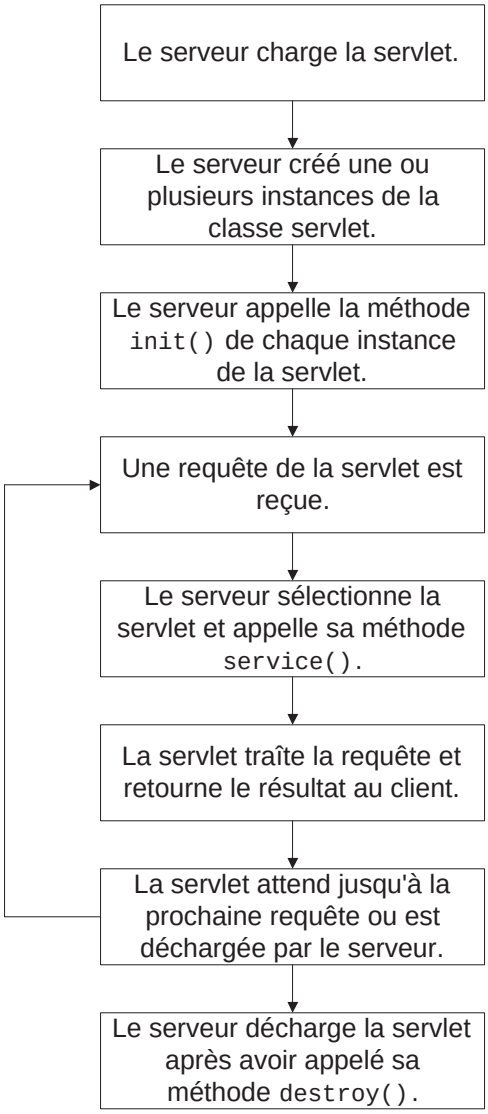

FIG. 3.1 – Cycle de vie d'une servlet

La performance est la différence la plus visible entre les servlets et les CGI. Puisque les servlets sont exécutées dans le même espace de processus que le serveur et qu'elles ne sont chargées qu'une seule fois, elles peuvent fournir une réponse plus rapide et plus efficace aux requêtes du client, contrairement aux CGI qui doivent créer un nouveau processus à chaque nouvelle requête. Le temps pris par la création du processus implique une baisse significative des performances. De plus, la création d'un seul processus permet de partager des ressources entre différentes requêtes. Par exemple, une connection à une base de données.

La portabilité est aussi un gros avantage des servlets sur les CGI. A l'inverse de la plupart des CGI, les servlets peuvent êtres exécutées sur différents serveurs et plateformes sans modification aucune. Cette caractéristique s'avère

très importante dans le développement d'applications largement distribuées.

Enfin, les servlets sont beaucoup plus sûres que les CGI. De la même manière que les applets s'exécutent dans un navigateur, les servlets non certifiées s'exécutent au sein d'une *sandbox*. Cette *sandbox* est un environnement dans lequel un programme ne peut pas accéder à des ressources extérieures comme le système de fichier ou le réseau. Bien sûr, ces restrictions peuvent êtres supprimées en fonction de la politique définie par le *Java Security Manager*.

Chapitre 4

CORBA

4.1 Introduction

L'*Object Management Group* (OMG) est un consortium international créé en 1989 et regroupant plus de 850 acteurs du monde informatique tels que IBM, Sun Microsystems, Hewlett-Packard, etc. L'objectif de ce groupe est de faire émerger des standards pour l'intégration d'applications distribuées hétérogènes à partir de technologies orientées objet. L'élément clé de l'OMG est CORBA [3] (*Common Request Broker Architecture*), une architecture d'objets distribués répondant aux besoins d'homogénéisation du développement et d'interopérabilité.

4.2 L'*Object Management Architecture* (OMA)

L'OMG a défini l'*Object Management Architecture Guide* qui a deux composants : le *Core Object Model* et l'OMA *Reference Architecture*.

4.2.1 Le *Core Object Model*

Le *Core Object Model* définit tous les concepts objets sur lesquels les spécifications CORBA sont construites. Il définit les bases d'un modèle objet et un environnement pour étendre ce modèle. Les concepts définis dans le *Core Object Model* sont :

- Les objets
- Les opérations, incluant leur signature, leurs paramètres et leurs valeurs de retour
- Les types non-objets
- Les interfaces

La spécification CORBA est un raffinement du *Core Object Model*.

4.2.2 OMA *Reference Architecture*

L'OMA *Reference Architecture* décrit un environnement pour l'intégration des applications distribuées (Figure 4.1).

Cette architecture contient :

ORB (*Object Request Broker*) C'est le bus de communication qui transmet les requêtes d'invocation sur les objets, indépendement de l'emplacement de ceux-ci sur le réseau. L'ORB est basé sur les principes définis dans le *Core Object Model*.

Application Objects ...

Object Services Ce sont des spécifications d'objets qui fournissent une assistance de bas niveau aux développeurs d'applications. Ils contiennent des services de nommage, notification d'évènements, management de transaction, etc. OGS (voir chapitre 5, page 27) est un de ces services.

Common Facilities ...

Domain Interfaces ...

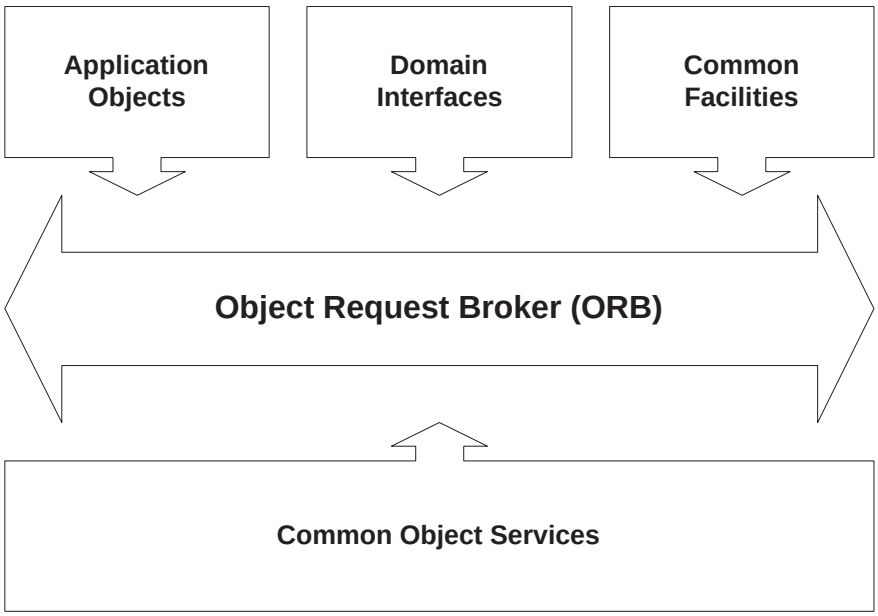

FIG. 4.1 – OMA *Reference Architecture*

4.3 *Interface Definition Language* (IDL)

Un des buts de CORBA est l'interopérabilité entre les différents systèmes. Pour cela, l'OMG a défini l'IDL qui permet de spécifier l'interface d'un objet quel que soit le langage utilisé pour implémenter cet objet.

Le tableau 4.1 contient les types de base du langage IDL avec leur équivalent dans le langage Java.

Type IDL	**Type Java**
boolean	*boolean*
char	*char*
double	*double*
float	*float*
long	*int*
long long	*long*
octet	*byte*
short	*short*
string	java.lang.String
unsigned *long*	*int*
unsigned *long long*	*long*
unsigned *short*	*short*
wchar	*char*
wstring	java.lang.String

TAB. 4.1 – Types de données IDL et leurs équivalents en Java

Des types de données plus complexes peuvent êtres décrits à l'aide des structures de données du tableau 4.2 :

Structure de donnée	**Description**
struct	Enregistrement (*record*).
enum	Ennumération.
union	Union discriminatoire.
sequence	Tableau borné ou non-borné.

TAB. 4.2 – Structures de données en IDL

4.4 Fonctionnement de CORBA

Une application CORBA est composée de deux entités principales : le client et le serveur. Le client est une application qui fait appels à des objets. Le serveur quand à lui contient l'implémentation de ces objets. Entre les deux, nous avons l'ORB qui se charge de passer les invocations du client au serveur quel que soient leur emplacement sur le réseau.

Entre le client et l'ORB, nous avons un objet appelé *stub* chargé de transformer les appels de méthode en messages. Pour ce faire, il doit tansformer les données de l'application (décrites en IDL) pour qu'elles puissent passer par l'ORB (*marshalling/unmarshalling*). Du côté du serveur, nous avons un objet similaire

appelé *skeleton* qui se charge de transformer les messages en appels de méthode.

Il existe plusieurs modes de fonctionnement pour les applications CORBA. Les principaux modes sont les modes statique et dynamique.

4.4.1 Mode statique

En mode statique, l'application cliente invoque une requête spécifique sur un objet. L'ORB se charge alors du transport de cette requête jusqu'à l'objet cible du côté serveur. La méthode appelée est alors activée ; son implémentation est recherchée pour pouvoir exécuter la requête proprement dite. Enfin, la réponse, si la requête en demandait une, est retournée à l'application cliente suivant le même principe.

Cette approche statique est bien adaptée pour la conception et l'exécution d'applications dont les spécifications IDL ne changent pas.

4.4.2 Mode dynamique

Le mode statique ne convient plus lorsque les interfaces des objets évoluent car il faudrait recompiler le client et le serveur à chaque modification. Pour améliorer ce mode d'invocation, CORBA offre un ensemble de mécanismes pour exploiter et implanter dynamiquement des objets répartis :

- l'interface d'invocation dynamique du côté du client (DII, *Dynamic Invocation Interface*)
- l'interface de squelette dynamique du coté du serveur (DSI, *Dynamic Skeleton Interface*)
- le référentiel des interfaces (IR, *Interface Repository*)

Grâce à ce mode dynamique, un client peut invoquer n'importe quelle opération sur n'importe quel objet. Pour cela, le client doit découvrir les informations relatives à l'interface des objets au moment de l'exécution de la requête. Il n'a pas besoin de connaître ces informations au moment de la compilation. C'est l'IR qui fournit ces données. Il est en quelque sorte la banque d'informations de tous les objets décrits en IDL.

Chapitre 5

Object Group Services (OGS)

OGS [?] est un service de communication de groupe pour CORBA. Il est composé d'un ensemble d'interfaces génériques spécifiées en IDL. Avec OGS, les clients peuvent envoyer des invocations à des groupes d'objets sans connaître le nombre et l'identité des membres du groupe. En plus, OGS fournit un support pour l'invocation de groupe transparente qui permet aux clients d'invoquer des opérations d'un groupe d'objets comme s'il s'agissait d'un objet simple.

5.1 Invocation avec OGS

La figure 5.1 illustre le fonctionnement de base de OGS. Les clients invoquent une méthode d'un objet en faisant appel à un groupe. OGS va se charger de transmettre cette requête à tous les membres du groupe et de collecter les réponses. Il retourne ensuite cette réponse au client qui a émis la requête.

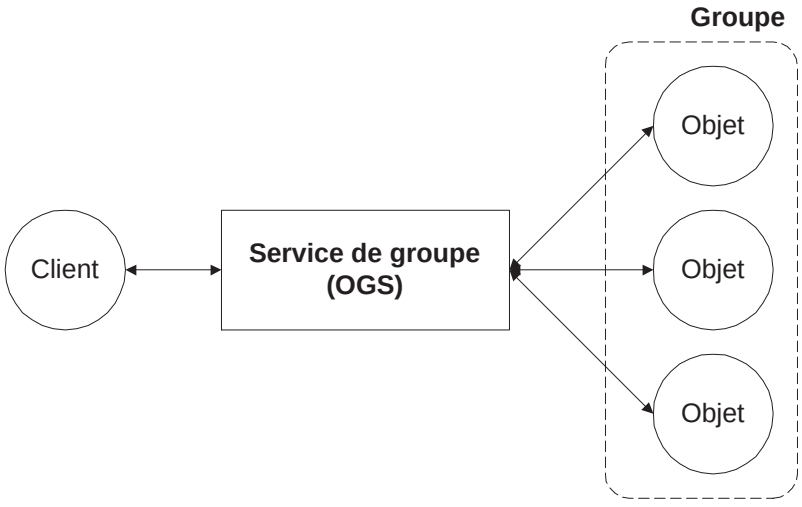

FIG. 5.1 – Invocation d'objet avec OGS

5.2 Changement de vue

Dans un groupe OGS, il se peut qu'un des membres tombe en panne et doive être retiré du groupe. Il se peut aussi qu'un nouvel objet veuille devenir membre de ce groupe. OGS va devoir faire ce qu'on appelle un changement de vue, une vue représentant l'ensemble des membres d'un groupe.

Dans le cas où un objet veut devenir membre du groupe, le changement de vue s'effectue comme suit :

- Le service de groupe va demander à tous les membres du groupe leur état actuel. Si ces états sont différents (normalement, ce n'est pas le cas), il va organiser un consensus pour décider quel état choisir.
- Le service de groupe va envoyer l'état choisi au nouvel objet et demande à celui-ci de se mettre dans cet état.
- Le nouvel objet est intégré dans le groupe et le service de groupe va définir une nouvelle vue.

Pour pouvoir faire ces opérations, tous les objets faisant partie ou voulant intégrer le groupe doivent implémenter les méthodes `get_state` et `set_state` qui permettent respectivement d'obtenir l'état de l'objet et de définir l'état de l'objet.

Dans le cas où un membre du groupe tombe en panne, le service de groupe va retirer l'objet défaillant et définir une nouvelle vue.

Chapitre 6

Architecture

Avant de considérer l'architecture de notre système de réplication de servlets dans son ensemble, nous allons examiner l'architecture d'une seule machine. Puis nous regarderons comment est intégrée la réplication.

6.1 Architecture sans réplication

La figure 6.1 nous montre l'architecture de notre système sur une seule machine (sans réplication).

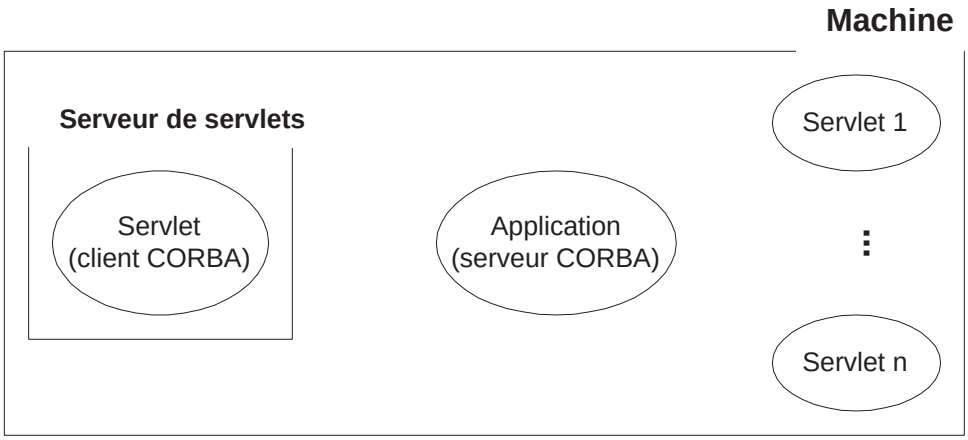

FIG. 6.1 – Architecture sur une seule machine

Le principal élément de cette architecture, c'est le serveur de servlet. C'est là que vont arriver les requêtes. Cela peut être un serveur HTTP qui fournit un service de servlets (Apache, Netscape-Enterprise Server, Java Web Server) ou un simple serveur de servlet (JSDK Servlet Runner, JRun, ServletExec).

Sur ce serveur de servlets, nous avons une servlet qui est en même temps un client CORBA. Le rôle de cette servlet est de transmettre toutes les requêtes qu'elle reçoit au serveur CORBA avec toutes les informations nécessaires (configuration du serveur de servlets). Elle se chargera aussi de retourner au client le

résultat de l'exécution de la requête par le serveur.

Le serveur CORBA est quand à lui une simple application Java. Son rôle est de recréer un environnement d'exécution de servlets, d'exécuter celles-ci en fonction des requêtes qui lui parviennent et de retourner au client CORBA le résultat de ces exécutions.

Les servlets qui se trouvent à droite de la figure 6.1 sont les servlets développées par l'utilisateur. On pourrait placer ces servlets directement sur le serveur de servlets et cela ne changerait rien à leur exécution.

Voyons maintenant en détail le traitement d'une requête. Les différents points se réfèrent à la figure 6.2

1. Le client HTTP envoie sa requête au client CORBA qui se trouve sur le serveur de servlets.
2. Le client CORBA crée un objet (décrit en IDL) et y place la requête et certaines informations sur la configuration du serveur de servlets. Puis il invoque le serveur CORBA avec cet objet en paramètre.
3. Le serveur CORBA analyse les informations qu'il a reçu du client CORBA et crée un environnement d'exécution de servlet approprié. Il analyse ensuite la requête et exécute la servlet demandée par le client HTTP.
4. Une fois l'exécution de la servlet terminée, celle-ci retourne une réponse (généralement du code HTML) au serveur CORBA.
5. Le serveur CORBA crée un objet (décrit en IDL) et y place la réponse de la servlet. Il retourne ensuite cet objet au client CORBA.
6. Le client CORBA envoie la réponse de la servlet au client HTTP.

6.2 Architecture avec réplication

L'architecture répliquée de notre système sera composée de plusieurs machines ayant toutes la même structure que celle que l'on a vue dans la section précédente. Pour permettre aux différents serveurs de communiquer entre eux, nous allons utiliser le service de groupe de CORBA (OGS).

Comme nous pouvons le voir sur la figure 6.3, tous les serveurs CORBA font partie du même groupe OGS.

Dans ce système, quand un client CORBA recevra une requête, il l'enverra au groupe OGS et la requête sera exécutée sur tous les serveurs qui font partie du groupe.

Le traitement d'une requête avec deux serveurs répliqués se déroulera de la manière suivante (voir figure 6.4) :

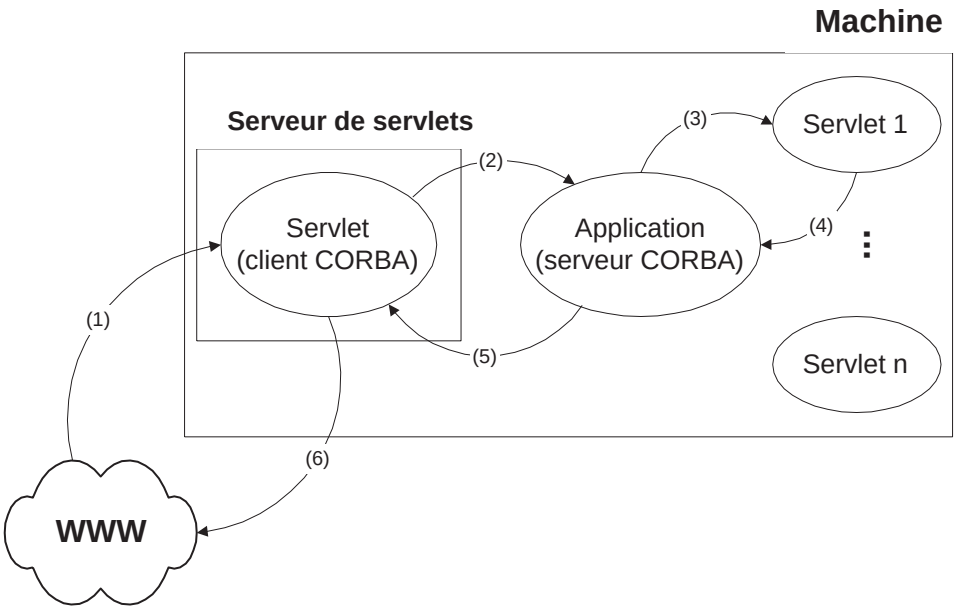

FIG. 6.2 – Exemple de requête

1. Le client HTTP envoi une requête au client CORBA qui se trouve sur le serveur de servlets.
2. Le client CORBA crée un objet (décrit en IDL) et y place la requête et certaines informations sur la configuration du serveur de servlets. Puis il invoque le groupe OGS avec cet objet en paramètre.
3. Le service de groupe OGS se charge de transmettre l'invoquation à tous les serveurs CORBA faisant partie du groupe.
4. Les serveur CORBA analysent les informations qu'ils ont reçu du client CORBA et créent chacun un environnement d'exécution de servlet approprié. Ils analysent ensuite la requête et exécutent la servlet demandée par le client HTTP.
5. Une fois l'exécution de la servlet terminée, celle-ci retourne une réponse (généralement du code HTML) au serveur CORBA.
6. Les serveurs CORBA créent un objet (décrit en IDL) et y placent la réponse de la servlet. Ils retournent ensuite cet objet au groupe.
7. Le service de groupe OGS se charge de recueillir les différentes réponses (identiques si tout a correctement fonctionné) et en envoie une au client CORBA qui à fait l'invoquation.
8. Le client CORBA envoie la réponse de la servlet au client HTTP.

6.3 Optimisation

Dans l'architecture telle que nous venons de la voir, toutes les requêtes sont répliquées sur tout les serveurs. Mais dans le protocole HTTP, il est bien spéci-

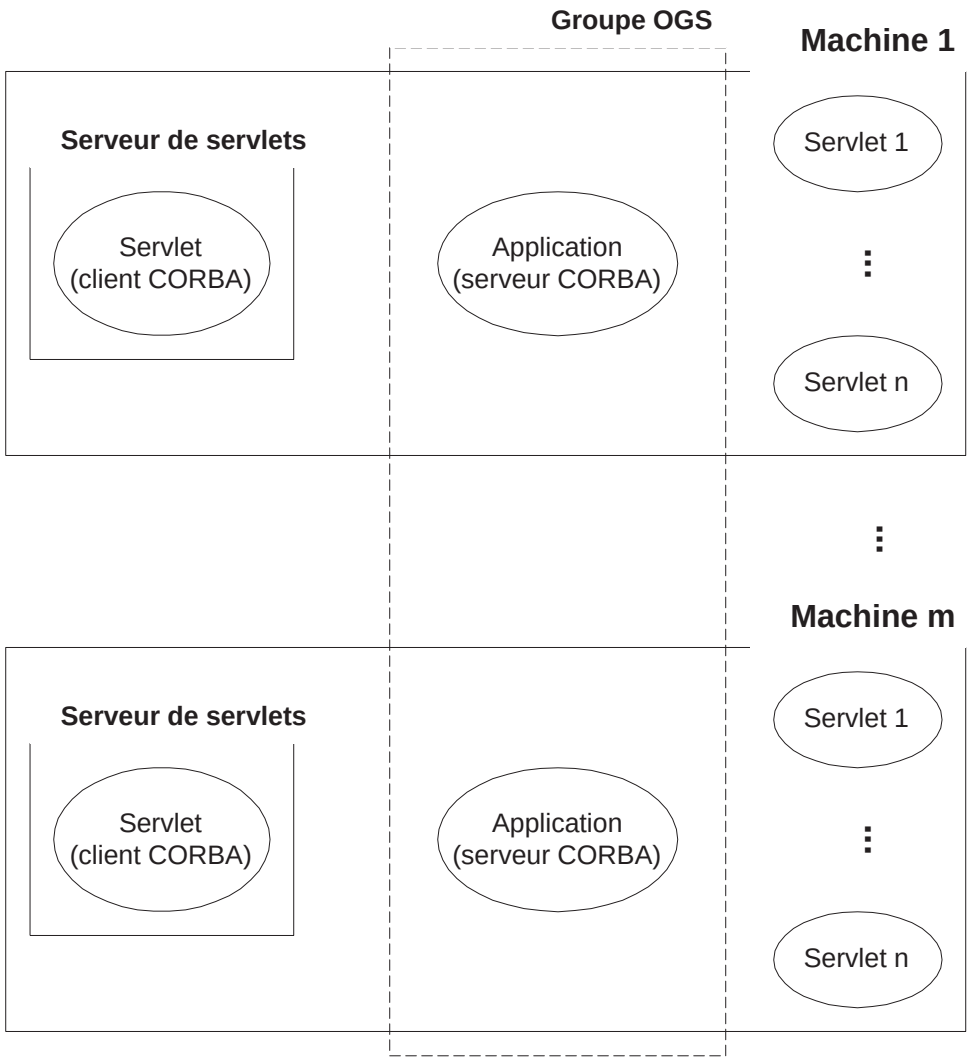

FIG. 6.3 – Architecture avec réplication

fié que les requêtes utilisant la méthode GET ne doivent en aucun cas modifier l'état du serveur. Il en est de même pour les requêtes utilisant la méthode HEAD.

Cela nous amène a penser qu'il n'est pas indispensable de répliquer ces requêtes. Nous allons donc les executer directement depuis le client CORBA. Cela nous evitera de passer par CORBA et OGS et devrait améliorer le temps de réponse des requêtes qui ne font que lire de l'information.

Avec cet optimisation, une requête en "lecture seule" se passera de la manière suivante (voir figure 6.5) :

1. Le client HTTP envoi une requête au client CORBA qui se trouve sur le serveur de servlets.

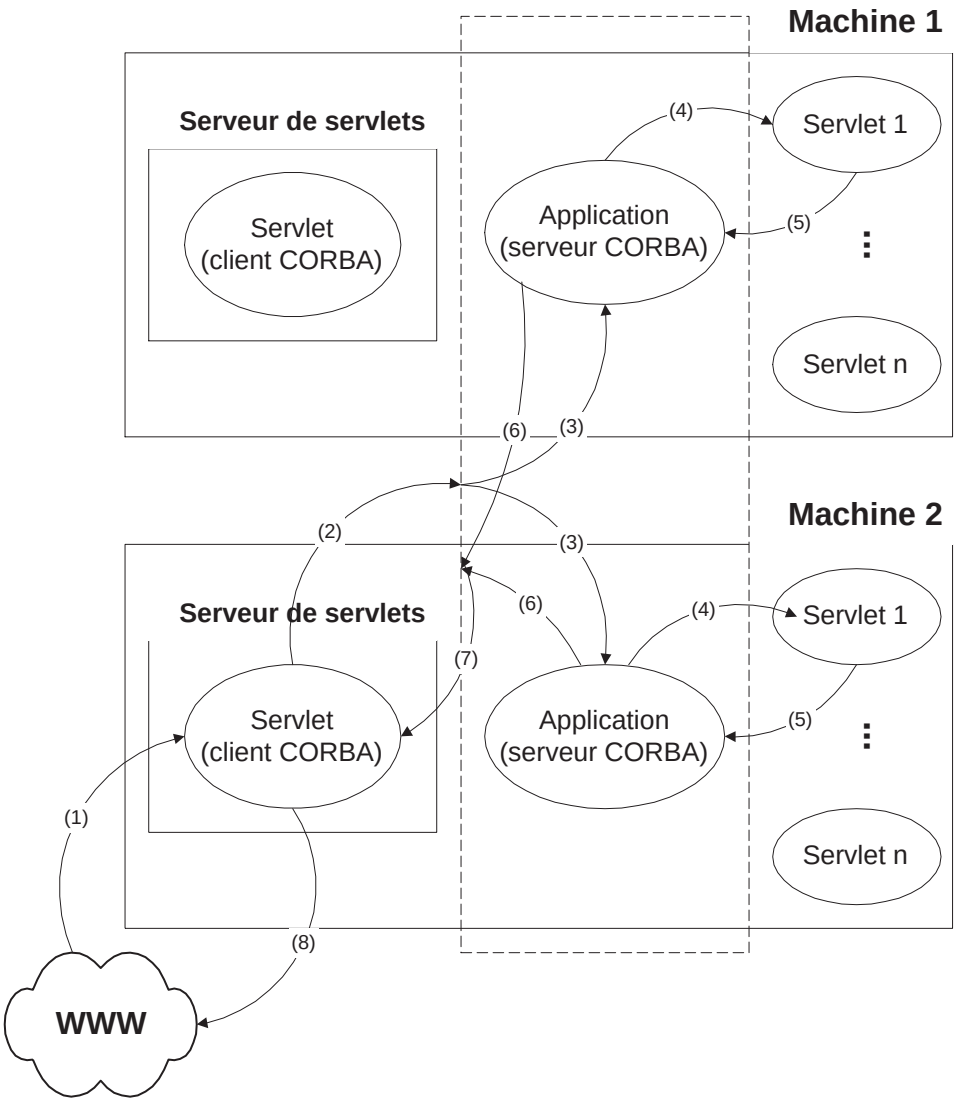

FIG. 6.4 – Exemple de requête avec réplication

2. Le client CORBA crée un environnement d'exécution de servlet approprié. Il analyse ensuite la requête et exécute la servlet demandée par le client HTTP.
3. Une fois l'exécution de la servlet terminée, celle-ci retourne une réponse (généralement du code HTML) au client CORBA.
4. Le client CORBA envoie la réponse de la servlet au client HTTP.

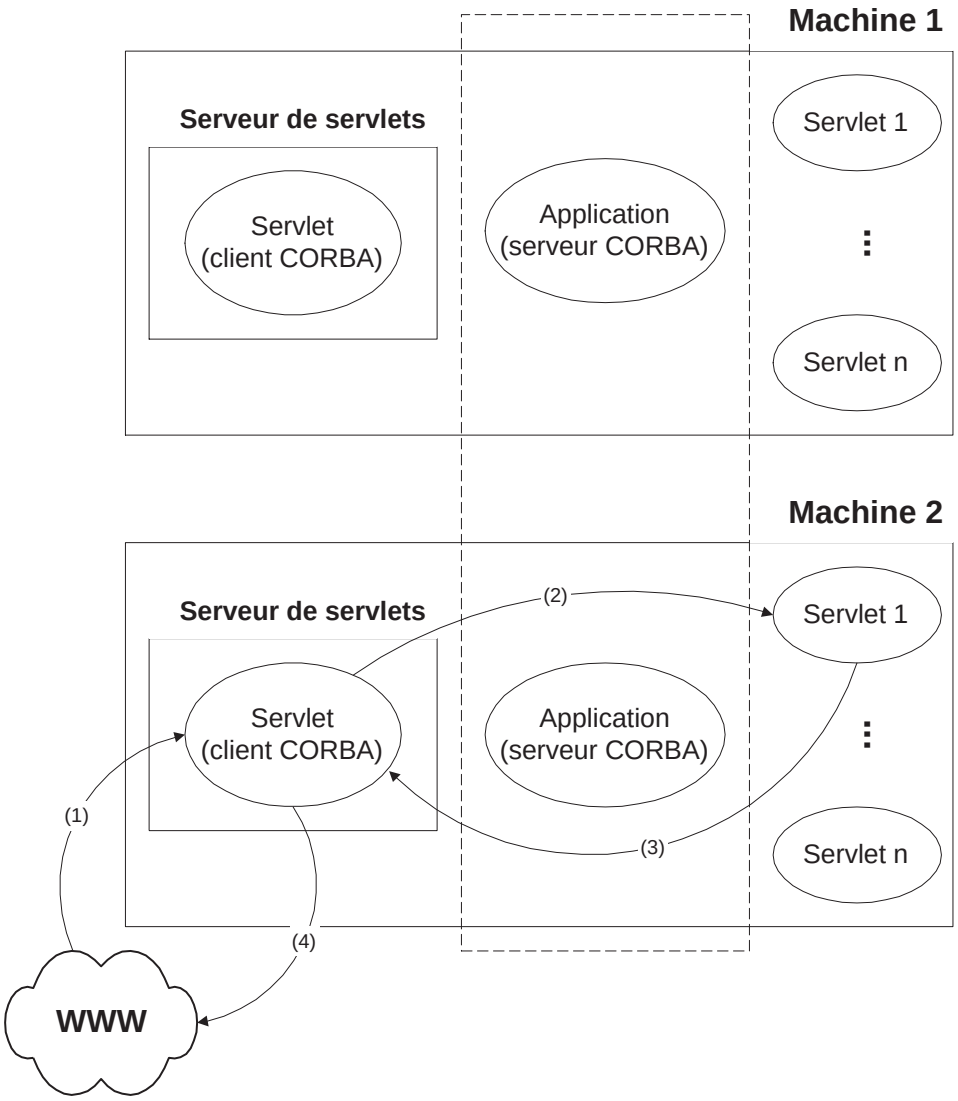

FIG. 6.5 – Exemple de requête en "lecture seule" avec réplication optimisée

Chapitre 7

Implémentation

L'implémentation d'une application utilisant le service de groupe de CORBA (OGS) se fait de la manière suivante. On développe l'application comme si elle ne devait pas être répliquée puis, une fois que celle-ci fonctionne correctement, on intègre le service de groupe (OGS) en ne modifiant que légèrement le code original.

Pour simplifier la compréhension, nous allons d'abord voir l'implémentation sans OGS, puis nous verrons ensuite les modifications que l'on a du apporter pour intégrer la réplication.

7.1 Implémentation sans réplication

L'implémentation de notre système peut être séparée en différentes parties :

- Les données IDL CORBA
- L'environnement d'exécution de servlets
- Le client CORBA
- Le serveur CORBA
- Les servlets de l'utilisateur

7.1.1 Les données IDL

Les données qui circulent entre le client et le serveur CORBA sont décrites à l'aide du langage de définition d'interfaces IDL. Nous appellerons cela les données IDL[1].

L'exécution d'une servlet nécessite trois objets des trois classes suivantes (API des servlets) :

- `ServletConfig`
- `HttpServletRequest`

[1] toutes les classes qui composent les données IDL font partie du package `Replication.ServletReplicatiionPackage`.

– `HttpServletResponse`

Le tableau 7.1 décrit ces trois classes et nous indique quelles sont leurs classes équivalentes en données IDL.

Classe	Description
`ServletConfig`	Objet contenant la configuration du serveur de servlet, nécessaire à l'initialisation de la servlet. Classe équivalente : `ServletConfigReplication`
`HttpServletRequest`	Objet contenant la requête du client HTTP destinée à la servlet. Classe équivalente : `HttpServletRequestReplication`
`HttpServletResponse`	Objet contenant la réponse fournie par l'exécution de la servlet. Classe équivalente : `HttpServletResponseReplication`

TAB. 7.1 – Description des données IDL

Toutes les informations contenues dans ces objets ne nous sont pas utiles. Néamoins, nous avons décrit une structure de données pouvant contenir un maximum d'informations. Cela nous permet de choisir plus simplement les données que nous voulons transmettre, sans devoir recompiler notre IDL à chaque modification.

La description des informations contenues dans les classes du tableau 7.1 ainsi que leur utilité dans notre système se trouvent dans l'annexe A (page 47).

Pour simplifier l'implémentation, deux classes supplémentaires ont été crées : `dataIn` et `dataOut` (voir figure 7.1).

Les objets de la classe `dataIn` contiennent les objets que l'on doit fournir à une servlet pour pouvoir l'exécuter, soit un objet de la classe `ServletConfigReplication` et un objet de la classe `HttpServletRequestReplication`.

Les objets de la classe `dataOut` contiennent l'objets qui est généré par l'exécution de la servlet soit un objet de la classe `HttpServletResponseReplication`.

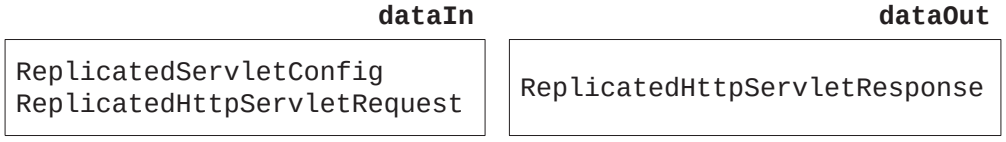

FIG. 7.1 – Classes `dataIn` et `dataOut`

7.1.2 L'environnement d'exécution de servlets

L'environnement d'exécution de servlets peut être séparé en deux parties : la réimplémentation des classes de l'API des servlets et le gestionnaire d'exécution de servlets.

Réimplémentation des classes de l'API des servlets

La réimplémentation des classes de l'API des servlets a pour but de créer les objets nécessaires à l'exécution d'une servlet (`ServletConfig`, `HttpServletRequest` et `HttpServletResponse`) à partir de données IDL.

Les objets de la classe `ServletConfig` sont utilisée lors de l'initialisation d'une servlet. C'est un objet de ce type qui est passé à la méthode `init()` de la servlet que l'on veut exécuter. La réimplémentation de cette classe s'appelle `ReplicatedServletConfig`. Son contsructeur prends en paramètre un objet de la classe `ServletConfigReplication` (données IDL).

Les objets de la classe `HttpServletRequest` contiennent toutes les informations sur la requête. Ils sont passés en paramètre à la méthode `service()` de la servlet que l'on veut exécuter. La réimplémentation de cette classe s'appelle `ReplicatedHttpServletRequest`. Son constructeur prends en paramètre un objet de la classe `HttpServletRequestReplication` (données IDL).

Les objets de la classe `HttpServletResponse` contiennent toutes les informations de la réponse résultant de l'exécution de la servlet. Ils sont passés en paramètre à la méthode `service()` de la servlet que l'on veut exécuter. La réimplémentation de cette classe s'appelle `RepliacatehttpServletResponse`. Son constructeur prends en paramètre un objet de la classe `HttpServletResponseReplication` (données IDL).

Gestionnaire d'exécution de servlets

Le gestionnaire d'exécution de servlets est une classe (`ReplicatedServer`) qui permet d'exécuter des servlets à partir de données IDL. Son comportement est identique à celui d'un serveur de servlets conventionel à la différence près que les requêtes ne lui arrivent pas de la même manière.

Attributs La classe `ReplicatedServer` contient un seul attribut. Il s'agit d'une table de hachage qui contient les servlets qui ont déjà été exécutées au moins une fois. Dans cette table, les clés sont les noms des servlets et les objets correspondents aux servlets elles-mêmes. Cette table permet de garder en mémoire les servlet pour ne pas avoir à les recharger et les réinitialiser à chaque appel d'une servlet.

Constructeur Le constructeur de la classe `ReplicatedServer` ne fait rien de particulier à part initialiser la table de hachage. Il ne prend aucun paramètre.

Méthodes La classe `ReplicatedServer` ne propose qu'une méthode :

```
public dataOut execute(dataIn inputData)
```

Cette méthode prends en paramètres un objet du type `dataIn` qui contient la configuration du serveur et la requête. En retour, elle fournit un objet `dataOut` qui contient la réponse.

Si la servlet demandée dans la requête passée en paramètres n'a jamais été exécutée, le fonctionnement de la méthode est le suivant (voir figure 7.2) :

1. La méthode reçoit en paramètres un objet de la classe `dataIn` (1).
2. On créé un nouvel objet de la classe `HttpServlet`.
3. On créé un objet de la classe `ReplicatedServletConfig` à partir de l'objet de la classe `dataIn` reçu en paramètres.
4. On initialise la nouvelle servlet en appelant sa méthode `init()` avec en paramètres l'objet créé au point précédent (2).
5. On créé un objet de la classe `ReplicatedHttpServletRequest` à partir de l'objet de la classe `dataIn` reçu en paramètres.
6. On exécute la servlet en appelant sa méthode `service()` avec en paramètres l'objet créé au point précédent (3).
7. On récupère la réponse de la servlet dans un objet de la classe `ReplicatedHttpServletResponse` (4).
8. On place la servlet dans la table de hachage.
9. On retourne un objet de la classe `dataOut` contenant la réponse de la servlet (5).

Si la servlet se trouve déjà dans la table de hachage, le comportement de la méthode sera le suivant (voir figure 7.2) :

1. La méthode reçoit en paramètres un objet de la classe `dataIn` (1).
2. On créé un objet de la classe `ReplicatedHttpServletRequest` à partir de l'objet de la classe `dataIn` reçu en paramètres.
3. On exécute la servlet en appelant sa méthode `service()` avec en paramètres l'objet créé au point précédent (3).
4. On récupère la réponse de la servlet dans un objet de la classe `ReplicatedHttpServletResponse` (4).
5. On met à jour la servlet dans la table de hachage.
6. On retourne un objet de la classe `dataOut` contenant la réponse de la servlet (5).

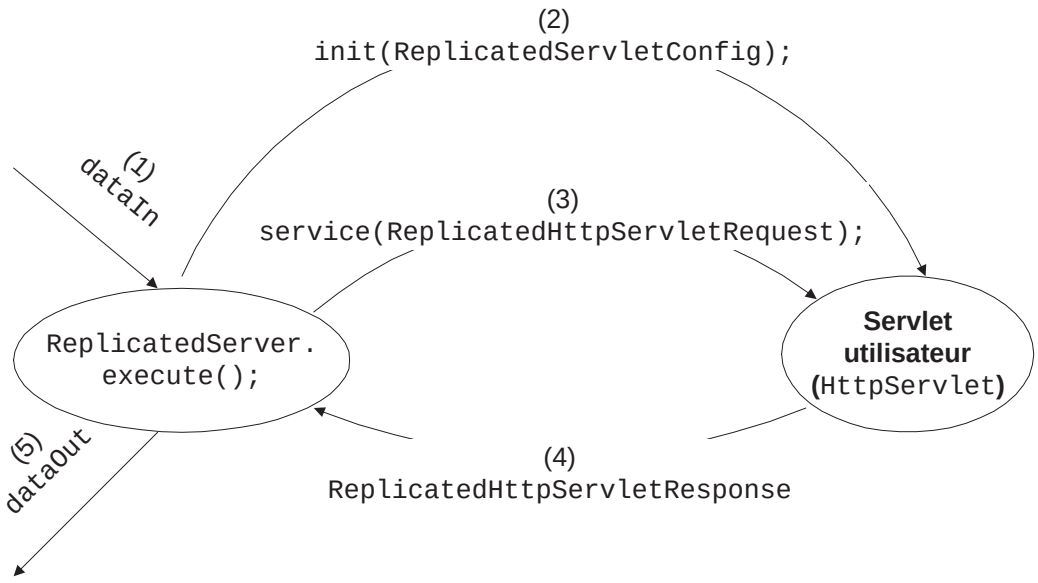

FIG. 7.2 – Exemple de fonctionnement de la méthode `execute()`

7.1.3 Le client CORBA

Le client CORBA est une servlet conventionnelle qui tourne sur le serveur de servlets. Cette servlet fonctionne de deux manières selon la méthode utilisée pour la requête.

Si la méthode utilisée est GET ou HEAD, la requête ne doit pas modifier l'état du serveur. Il n'y a donc pas de raison de répliquer la requête. Le client CORBA se charge alors lui-même d'exécuter la servlet demandée dans la requête à l'aide d'un objet de la classe `ReplicatedServer`.

Si la méthode de la requête est autre que GET ou HEAD, la requête est susceptible de modifier l'état du serveur. Il faut donc envoyer la requête à tous les serveurs pour que la servlet demandée soit exécutée par tous.

Dans les deux cas, le client CORBA doit récupérer la réponse fournie par la servlet et les transmettre au client HTTP.

7.1.4 Le serveur CORBA

Le serveur CORBA est une simple application Java qui reçoit les requêtes sous forme de données IDL (objet de la classe `dataIn`) et qui en fonction de ces données exécute une servlet à l'aide d'un gestionnaire d'exécution de servlets (objet de la classe `ReplicatedServer`). Une fois la servlet exécutée, le serveur CORBA envoies le résultat de l'exécution de la servlet (objet de la classe `dataOut`) au client CORBA.

7.1.5 Les servlets de l'utilisateur

Les servlets de l'utilisateur sont des servlets conventionnelles. Ce sont elles qui sont exécutées en fonction des requêtes des clients HTTP.

7.2 Implémentation avec réplication (OGS)

Expliquer comment chaque servlet passe son état et comment le service de réplication de servlet prend tous les états pour le passer aux autres répliquas.

+ multicast et deliver...

Chapitre 8

Mise en oeuvre

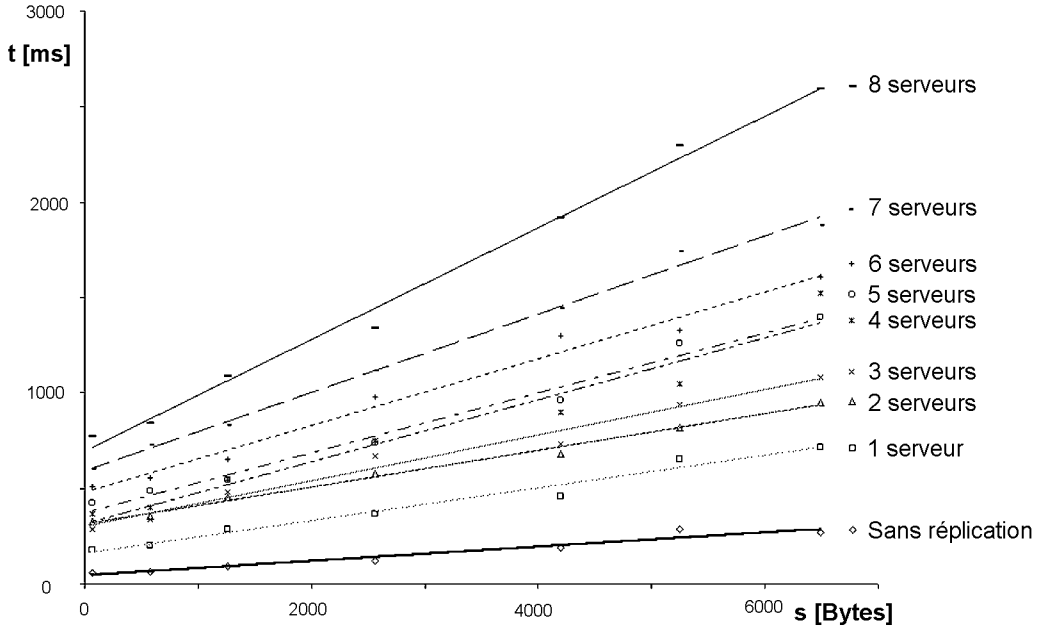

FIG. 8.1 – Graphique du temps d'exécution d'une requête (écriture) en fonction de la taille des données de celle-ci

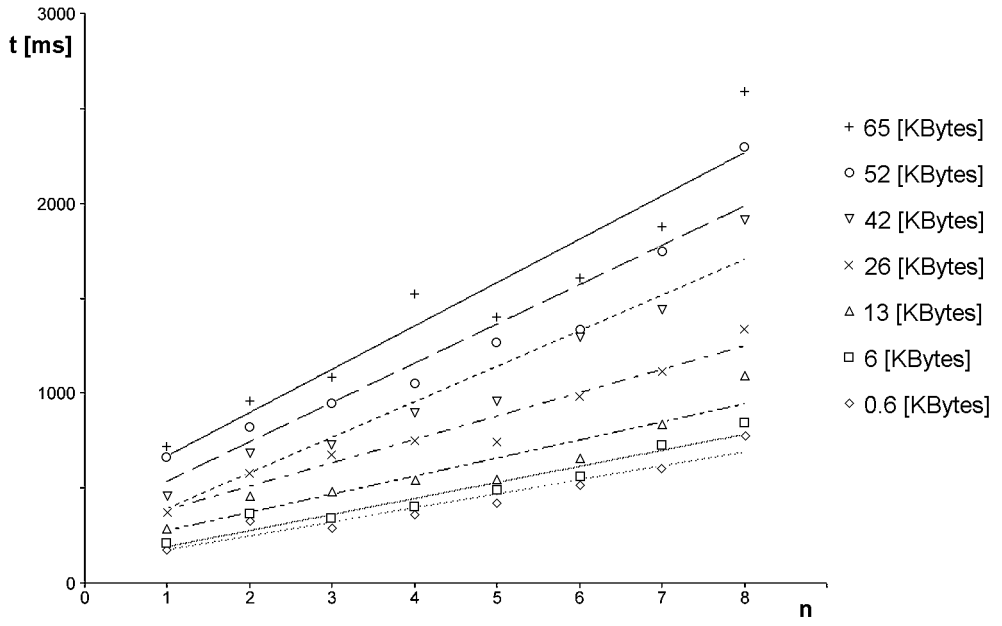

FIG. 8.2 – Graphique du temps d'exécution d'une requête (écriture) en fonction du nombre de serveurs

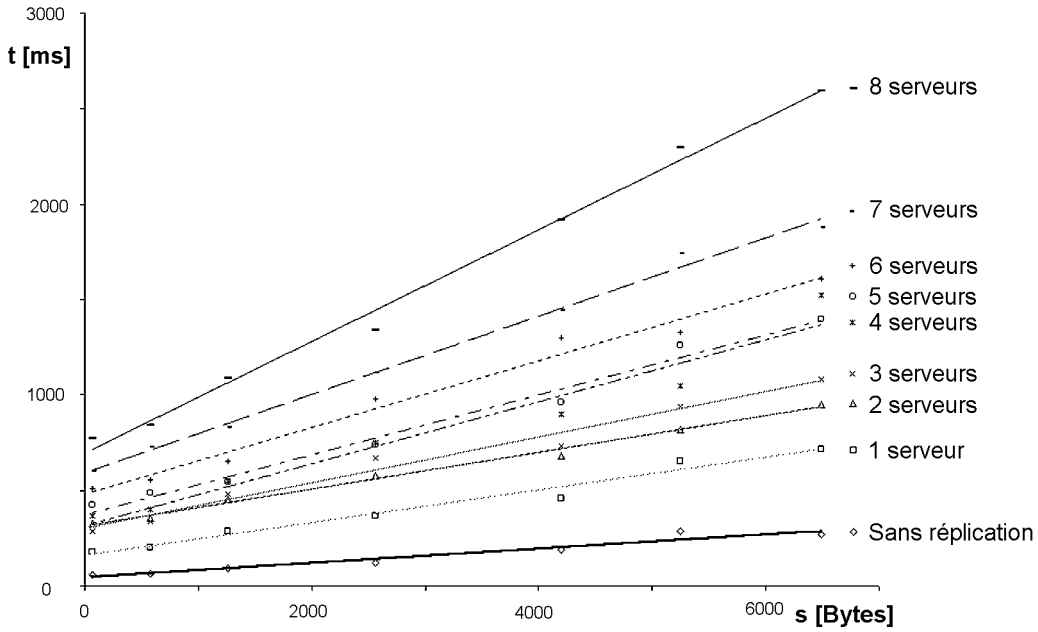

FIG. 8.3 – Graphique du temps d'exécution d'une requête (lecture seule) en fonction de la taille des données de celle-ci

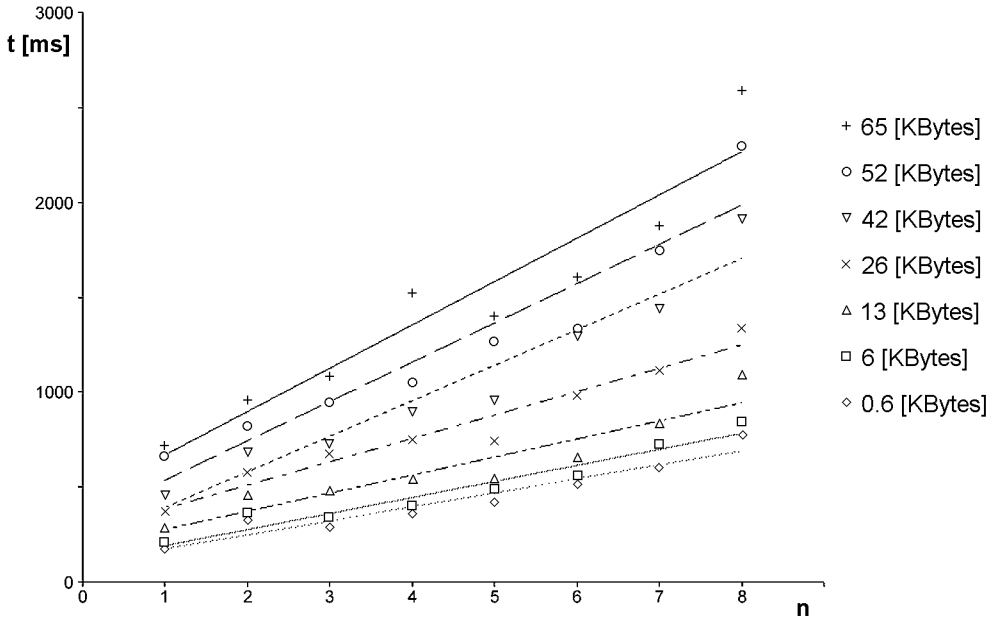

FIG. 8.4 – Graphique du temps d'exécution d'une requête (lecture seule) en fonction du nombre de serveurs

Chapitre 9

Conclusion

9.1 Etat du projet

...

9.2 Améliorations possibles

...

9.3 Leçons apprises durant ce projet

...

Annexe A

Données IDL

A.1 Classe `ServletConfig`

Paramètres d'initialisation

Les méthodes `getInitParameter()` et `getInitParameterNames()` retournent les paramètres d'initialisation de la servlet. Ces paramètres sont lus dans un fichier. Cela permet au developpeur de changer certains paramètres d'exécution de la servlet sans devoir redémarrer le serveur de servlets.

Ces paramètres ne nous intéressent pas pour notre système. Chaque serveur répliqué aura son propre fichier contenant les paramètres d'initialisation. Nous n'allons donc pas les intégrer dans les données IDL.

Contexte d'exécution

La méthode `getServletContexte()` retourne un objet de la classe `ServletContext` qui contient toutes les informations sur le contexte d'exécution de la servlet.

Certaines informations de cette classe nous serons utiles. Nous allons donc détailler cette classe dans la section A.2 et voir lesquelles nous intégrerons dans les données IDL.

A.2 Classe `ServletContext`

Attributs

Les méthodes `getAttribute()` et `getAttributeNames()` retournent les attributs de la servlet. Ce sont des objets qui nous fournissent des informations supplémentaires à propos du serveur de servlet.

Ces informations peuvent nous être utile. Nous allons donc les intégrer dans les données IDL. Seulement, il faut que les attributs soient sérializables. Si un attribut ne l'est pas, il ne sera pas intégré.

Version de l'API

Les méthodes `getMajorVersion()` et `getMinorVersion()` nous fournissent la version de l'API des servlets qui est supportée par le serveur.

Cette information ne nous sera pas utile, car notre environnement d'exécution de servlets aura sa propre version. Un changement de serveur de servlets ne devrait impliquer aucune modification de notre environnement d'exécution de servlet.

Informations sur le serveur de servlet

La méthode `getSreverInfo()` retourne le nom et la version du serveur de servlet.

Notre environnement d'exécution de servlet étant totalement indépendant du serveur de servlets, nous n'utiliserons pas cette information. Notre environnement aura son propre nom et sa propre version.

A.3 Classe `HttpServletRequest`

Cette classe est une sous-classe de la classe `ServletRequest`. Cela implique qu'une grande partie des méthodes présentées ci-dessous sont héritées de la classe `ServletRequest`.

Attributs

Les méthodes `getAttribute()` et `getAttributeNames()` retournent les attributs de la requête. Ce sont des objets qui fournissent des informations supplémentaires sur la requête.

Ces informations peuvent nous être utiles. Nous allons donc les intégrer dans les données IDL. Comme pour les attributs de la classe `SetvletContext`, ces objets devront être sérialisables pour pouvoir être intégrés.

Encodage des caractères

La méthode `getCharacterEncoding` retourne l'encodage utilisé pour des données du corps de la requête.

Cette information est indispensable et sera intégrée dans les données IDL.

Taille et type des données du corps de la requête

Les méthodes `getContentLength` et `getContentType` fournissent respectivement la taille et le type des données contenues dans le corps de la requête.

Ces informations sont indispensables et seront intégrées dans les données IDL.

Corps de la requête

Les méthodes `getInputStream()` ou `getReader()` nous donnent accès aux données contenues dans le corps de la requête.

Ces données sont évidement indispensables et seront intégrées dans les données IDL.

Paramètres

Les méthodes `getParameter()`, `getParameterNames()` et `getParameterValues` nous fournissent les paramètres contenus soit dans la *query string* soit dans le corps de la requête. Ces paramètres viennent généralement d'un formulaire HTML.

Comme nous allons inclure dans les données IDL le contenu du corps de la requête et la *query string*, il n'est pas nécessaire d'y intégrer en plus les paramètres (redondance).

Protocole

La méthode `getProtocol()` retourne le protocole utilisé pour la requête.

Cette information doit être intégrée dans les données IDL.

Adresse et *host* du client

Les méthodes `getRemoteAddr()` et `getRemoteHost()` nous fournissent respectivement l'adresse IP et le nom du *host* du client qui à envoyé la requête.

Ces informations doivent être intégrée dans les données IDL.

Scheme

La méthode `getScheme()` retourne le *scheme* de la requête. Par exemple, dans une requête HTTP, le *scheme* sera http.

Cet information doit être intégrée dans les données IDL.

Nom du serveur

La méthode `getServerName()` retourne le nom du serveur qui à reçu la requête.

Dans notre système de réplication, nous voulons que chaque serveur traite une requête comme si c'est lui qui avait reçu la requête. Nous n'intégrerons donc pas cette information dans les données IDL.

Port de réception

La méthode `getServerPort()` retourne le numéro du port sur lequel la requête a été reçue.

Cette information ne nous sera pas utile étant donné que notre environnement d'exécution de servlet ne reçoit pas vraiment les requêtes sur un port. De plus, tous les serveurs de servlets ne recevront pas forcement les requêtes sur le même port. Nous n'intégrerons donc pas cette information dans les données IDL.

Plan d'authentification

La méthode `getAuthType()` retourne le plan d'authentifiction de la requête.

Cette information sera intégrée dans les données IDL.

Cookies

La méthode `getCookies()` retourne l'ensemble des *cookies* contenus dans la requête.

Ces *cookies* seront intégrés dans les données IDL.

Entêtes

Les méthodes `getHeader()` et `getHeaderNames()` fournissent les entêtes contenues dans la requête HTTP.

Ces entêtes seront intégrées dans les données IDL.

Méthode

La méthode `getMethod()` retourne la méthode utilisée dans la requête HTTP.

Cette information sera intégrée dans les données IDL.

Path information

La méthode `getPathInfo()` retourne la chaîne de caractères qui suit le nom de la servlet invoquée dans l'URL de la requête.

Cette information sera intégrée dans les données IDL avec une petite modification. Nous allons supprimer le nom de la servlet qui à reçu la requête (le client CORBA) de l'URL.

Path translated

La méthode `getPathTranslated()` retourne la même chaîne de caractères que la méthode précédente (`getPathInfo()`), mais convertie en *path* correspondant à la machine qui à reçu la requête.

Cette information ne sera pas intégrée dans les données IDL car tous les serveurs répliqués n'auront pas forcement la même organisation des fichiers.

query string

La méthode `getQueryString()` retourne la *query string* de la requête.

Cette information sera intégrée dans les données IDL.

Nom du client

La méthode `getRemoteUSer()` retourne le nom du client qui à fait la requête.

Cette information sera intégrée dans les données IDL.

Identificateur de la session HTTP

La méthode `getRequestedSessionId()` retourne l'identificateur de la session HTTP associée à la requête.

Cette information sera intégrée dans les données IDL.

URI

La méthode `getRequestURI()` retourne une partie de l'URL de la requête HTTP sans la *query string*. Par exemple, pour l'URL /servlet/calcul ?a=12&b=24&op=mul, la méthode retournera /servlet/calcul.

Cette information sera intégrée dans les données IDL avec une petite modification. Nous allons supprimer le nom de la servlet qui à reçu la requête (le client CORBA) de l'URL.

Path de la servlet

La méthode `getServletPath()` retourne une partie de l'URL de la requête HTTP qui identifie la servlet invoquée. Par exemple, pour une servlet correspondant à l'URL /servlet/calcul qui reçoit une requête faite avec l'URL /servlet/Calcul/entiers/addition, la méthode retournera /servlet/Calcul.

Cette information sera intégrée dans les données IDL avec une petite modification. Nous allons supprimer le nom de la servlet qui à reçu la requête (le client CORBA) de l'URL.

Session HTTP

La méthode `getSession()` retourne un objet de la classe `HttpSession` qui contient toutes les informations concernant la session HTTP liée à la requête.

Certaines informations de cette classe nous seront utiles. Nous allons donc détailler cette classe dans la section A.4 et voir lesquelles nous intégrerons dans les données IDL.

Validité de l'identificateur de la session HTTP

La méthode `isRequestedSessionIdValid()` retourne un booléen indiquant si la requête correspond à une session valide.

Cette information sera intégrée dans les données IDL.

Provenance de l'identificateur de la session HTTP

Les méthodes `isRequestedSessionIdFromCookie()` et `isRequestedSessionIdFromURL()` retournent les deux un booléen qui indiquent si la requête a été envoyée par le client comme un *cookie* ou comme une partie d'URL.

Ces informations seront intégrées dans les données IDL.

A.4 Classe `HttpSession`

Date de création de la session

La méthode `getCreationTime()` retourne un entier long qui correspond au temps (en millisecondes) écoulé entre le 1er janvier 1970 et le moment de la création de la session.

Cette information sera intégrée dans les données IDL.

Identificateur de la session

La méthode `getId()` retourne l'identificateur de la session.

Cette information sera intégrée dans les données IDL.

Date du servier accès

La méthode `getLastAccessedTime()` retourne un entier long qui correspond au temps (en millisecondes) écoulé entre le 1er janvier 1970 et le moment de la dernière requête ayant utilisé le même identificateur que la session courante.

Cette information sera intégrée dans les données IDL.

Temps maximum de maintient de la session

...

Valeurs

...

Nouvelle session

...

A.5 Classe `HttpServletResponse`

Encodage des caractères

...

Corps de la réponse

...

Taille et type des données du corps de la réponse

...

Cookies

...

Erreurs

...

Redirection

...

Entêtes

...

Status

...

Bibliographie

[1] Dustin R. Callaway. *Inside Servlets (Server-Side Programming for the Java Platform)*. Addison Wesley, 1999.

[2] Pascal Felber. *The CORBA Object Group Service (A Service Approach to Object Groups in CORBA)*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, 1998. Number 1867.

[3] Reaz Hoque. *CORBA for Real Programmers*. Morgan Kaufmann, 1999.

[4] Suzanne Ahmed James Duncan Davidson. *Java Servlet API Specification - Version 2.1a*. Java Software Division, 1998. http ://java.sun.com/products/servlet/2.1/servlet-2.1.pdf.

[5] J. Gettys J. Mogul DEC H.Frystyk R. Fielding, UC. Irvine and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. Number RFC 2068. MIT/LCS, January 1997.

[6] R. Fielding T. Berners-Lee and H.Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. Number RFC 1945. MIT/LCS, May 1996.